CS450

Structure of Higher Level Languages

Lecture 21: What is a PhD? / Pattern matching

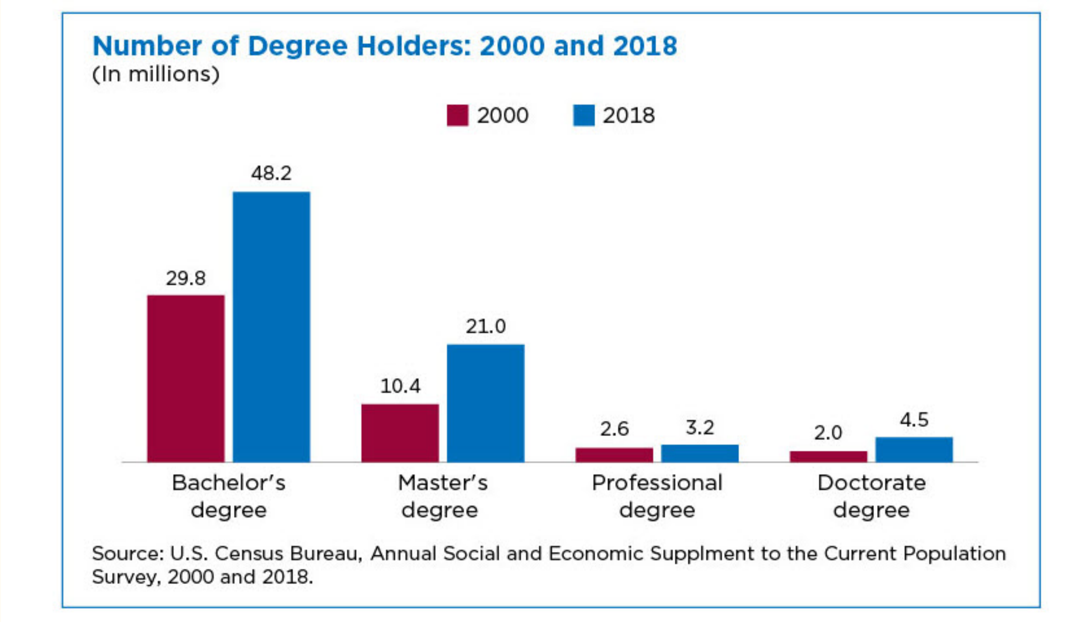
Tiago Cogumbreiro

What is a Ph.D.?

What is a Ph.D.?

> An academic degree where you must:
>
> 1. Master a subject completely
> 2. Advance the state of the art

- **Meaning:** Doctor of Philosophy
- **Importance:** The highest academic degree
- **Rarity:** Specialized workforce (4.5% of the population)
- **Prestige:** The title of Doctor



Number of Degree Holders: 2000 and 2018

(In millions)

Source: U.S. Census Bureau, Annual Social and Economic Supplment to the Current Population Survey, 2000 and 2018.

Source: `www.cs.purdue.edu/homes/dec/essay.phd.html`

Overview: What is a Ph.D.?

1. Why join graduate school?
2. Why not join graduate school?
3. Why a graduate degree in CS?
4. What is the structure of a PhD?
5. How do the a PhD effectively?

Why join graduate school?

Why join graduate school?

- **Intellectual curiosity:** the challenge of learning, the culture of seeking and ***sharing*** knowledge
- **Specialized degree:** after graduation you will be a better professional
- **Autonomy:** you want time to develop your own project
- **Better paying work prospects:** a graduate degree is a good investment

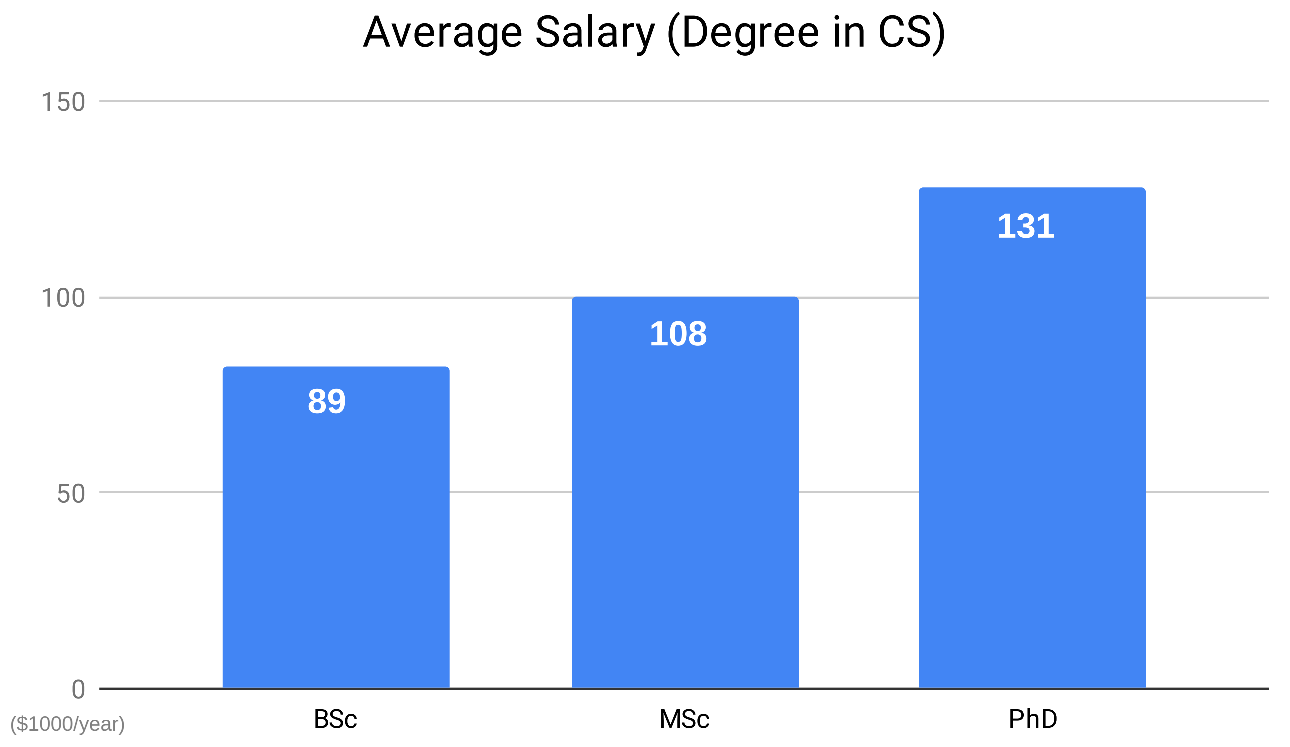
PhD degrees are generally fully-funded!

Why not join graduate school?

- **5-year investment:** You will not be paying tuition, grants and serving as a teaching assistant (TA) will pay you a stipend.
  However, this stipend is significantly lower than working in the industry!
- **Higher workload:** Graduate course are more rigorous than undergraduate courses. You will need to juggle TA with courses and research.
- **5-year commitment:** You will be working on the ***same*** subject for 5 years.
- **Autonomy required:** A PhD degree is not structured like a BSc. There is no exact formula for an effective PhD degree. More freedom, more responsibility.
- **Traveling required:** You will need to travel internationally.
- **Public speaking:** A crucial part of the PhD is public speaking.

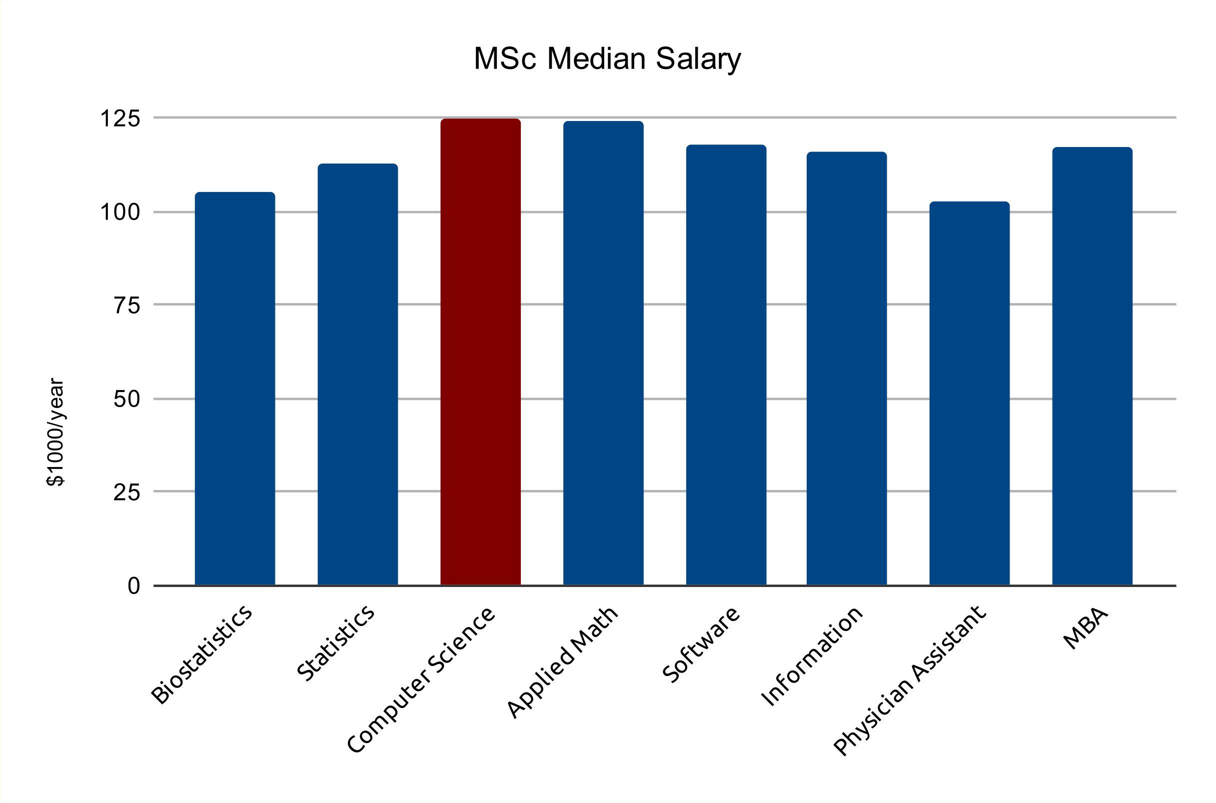
> I am using 5 years as an approximate duration to conclude a PhD degree.

Why join graduate school?



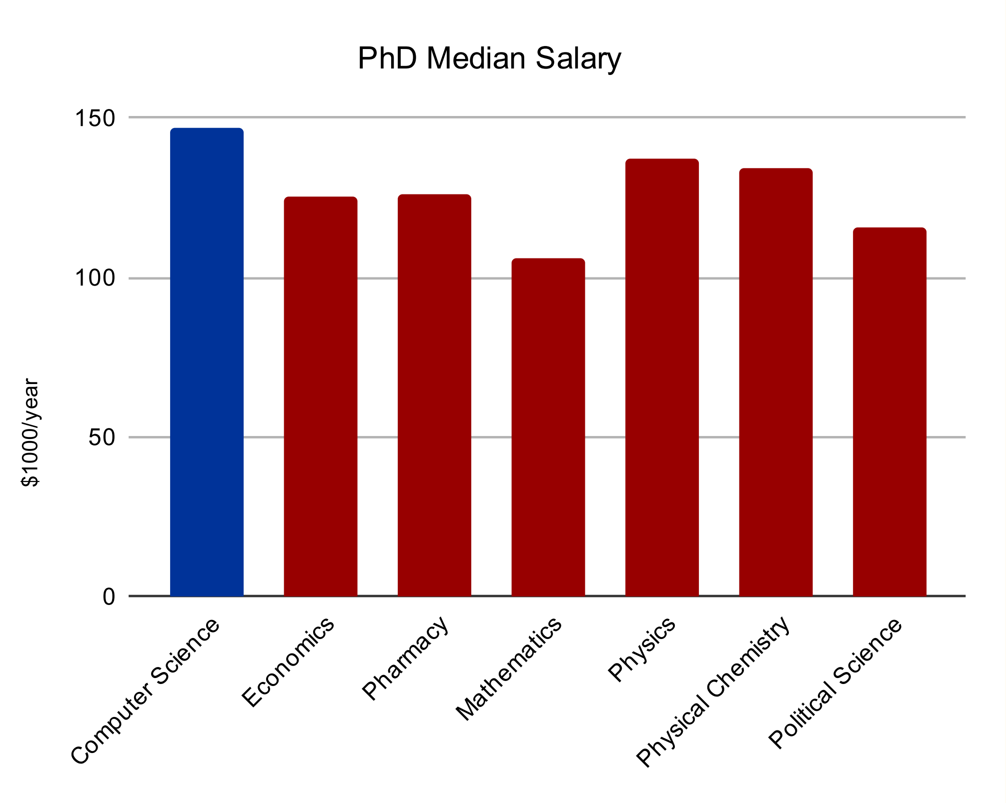
Sources: Payscale.com 2022 [1] [2] [3]

Why a graduate degree in CS?



Source: Best And Worst Graduate Degrees For Jobs in 2016. Lydia Dishman. Fortune, 2016.

Why a graduate degree in CS?



Source: Best And Worst Graduate Degrees For Jobs in 2016. Lydia Dishman. Fortune, 2016.

During your Ph.D. you must:

1. Master a subject completely
2. Advance the state of the art

The PhD degree

1. How to master a subject?

- Take **graduate courses**
- **Read** the literature: peer-reviewed scientific papers, books
- Attend **conferences**: meet top experts
- Attend **summer schools**: learn from world-class scholars
- Visit universities
- Do internships

> What are peer-reviewed papers? Scientific articles are submitted to other scientists experts in the field, who attest the scientific accuracy of the article. Articles may also be presented in a conference.

The PhD degree

2. How to advance the state of the art?

Complete a PhD thesis manuscript

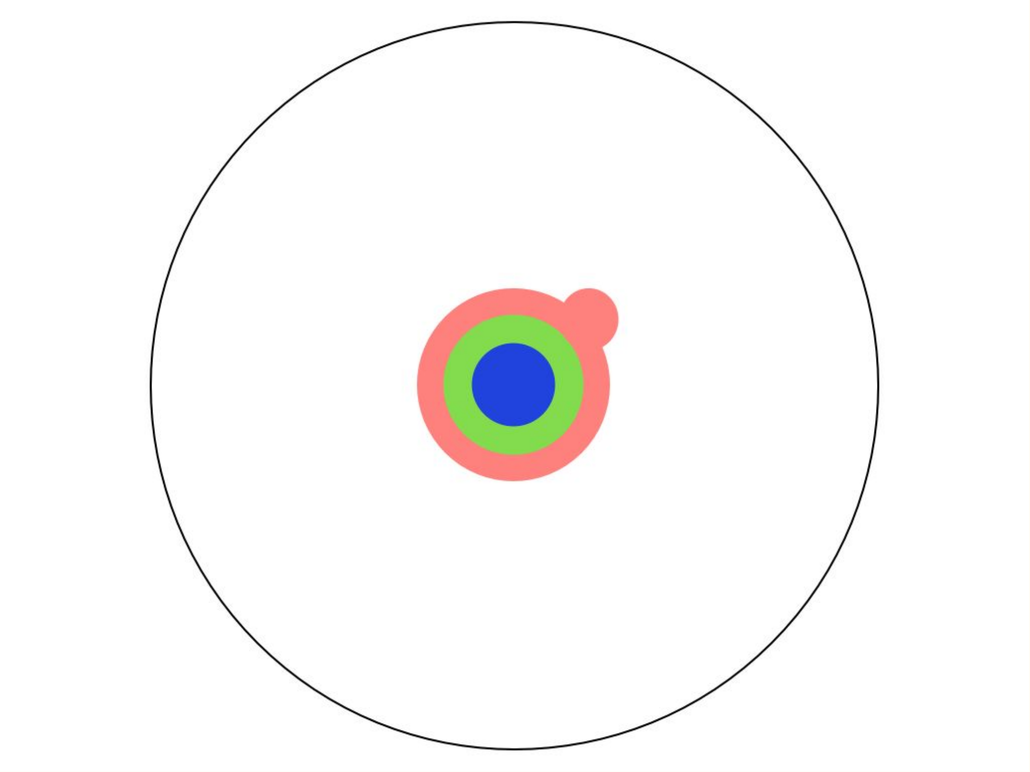
- **Novel:** the contribution must be completely new
- **Impact:** the contribution must have a useful impact to society

Skills

- explore, investigate, contemplate
- conceptualize, find issues, solve problems

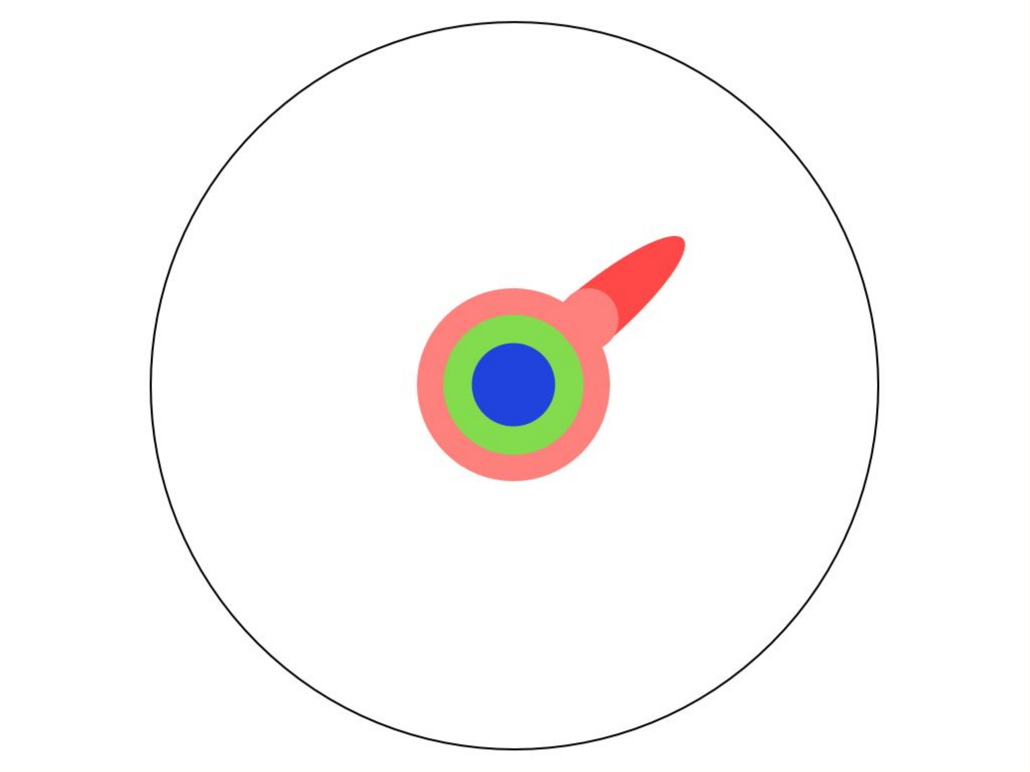
You will be the **world expert** on a subject!

Let us say you are here



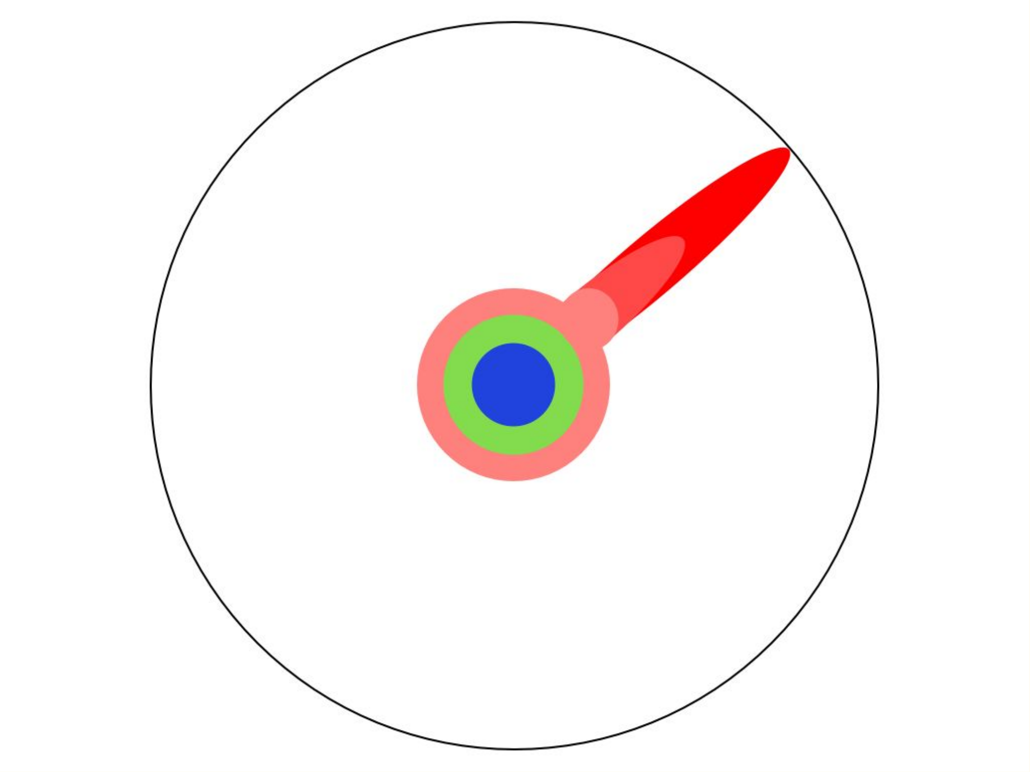
Source: matt.might.net/articles/phd-school-in-pictures/

Step 1: complete PhD courses (MSc)



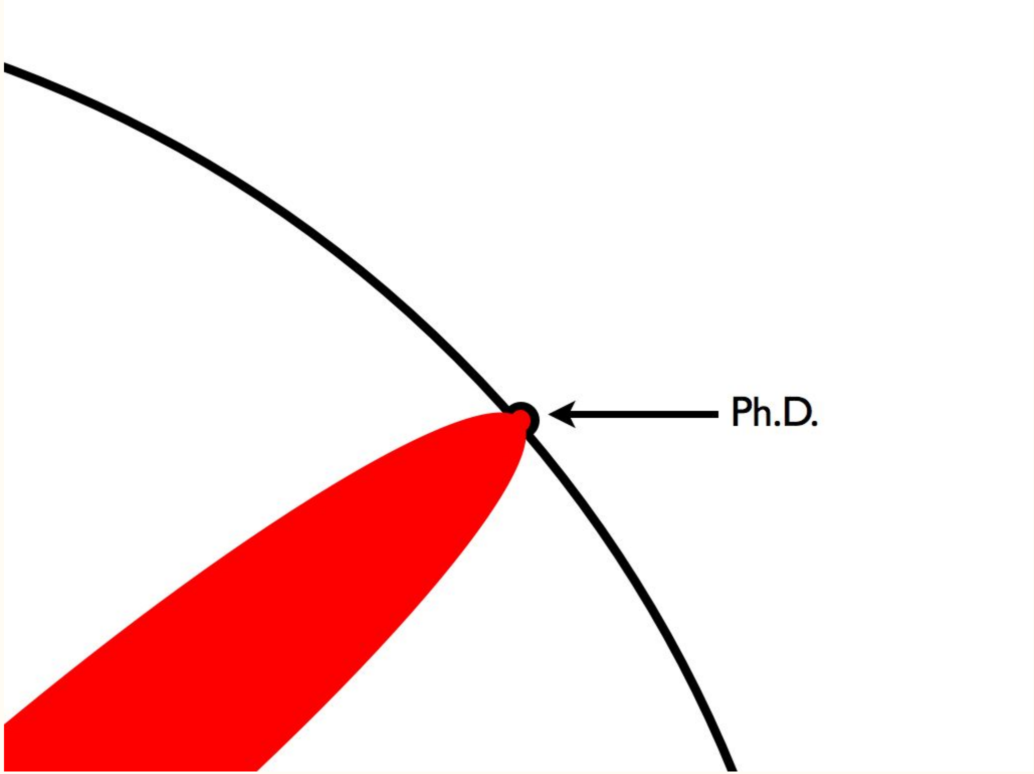
Source: `matt.might.net/articles/phd-school-in-pictures/`

Step 2: master a subject completely



Source: matt.might.net/articles/phd-school-in-pictures/

Step 3: advance the state of the art



Source: matt.might.net/articles/phd-school-in-pictures/

Pursuing a Ph.D. effectively

A PhD adviser shall...

- **Advise the student.** Help find a thesis topic, teach how to do research, write papers, give talks, etc.
- **Protect the student.** Provide protection from and information about funding concerns.
- **Inform the student.** Proactively provide realistic, honest advice about post-Ph.D. career prospects.
- **Frame student's work.** Provide early and clear guidance about the time frames and conditions for graduation.

A PhD student shall...

- **Get educated about career prospects post-PhD.**
- **Determine if these career prospects match your expectations.**
- **A PhD is not just research.** There is coursework, quals, and writing a thesis.
- **Work hard and maintain a rhythm.**
- **Follow the PhD program.** You are responsible for meeting the program's deadlines and requirements.

Research in the Software Verification Lab

Software Verification Lab

We make your programs run right

- We study how systems work
- We describe what we learned mathematically
- We understand why systems fail
- **We build tools** that help programmers

Members

- ***Professors***: Tiago Cogumbreiro, Julien Lange
- ***PhD***: Dennis Liew, Greg Blike, Hannah Zicarelli, Paul Maynard
- ***MS***: Ramsey Harrison
- ***BS***: Austin Guiney, Kleopatra Gjini (BS Thesis), Udaya Sathiyamoorthy (Ind. Study)

Software Verification Lab

The big picture

- We care about High Performance Computing
  (the backbone of scientific advancement)
- We focus on large-scale scientific workloads
- Our research improves the quality assurance of scientific codes

Looking for collaborators

- Summer/winter research projects

> Check out the more than 40 software open source projects, written in Python, C++, Java, OCaml, Coq, Racket, …

What you will learn...

Intersection between

- Software Engineering
- Logic

Things you may learn

- Functional programming
- Multithreading/parallel programming
- Developing Continuous Integration pipelines
- Using super computers (clusters in national labs with 1000s of cores)
- Implementing compilers/interpreters/debuggers
- Programming proofs & proof engineering
- Using SAT/SMT solvers & model checkers

Pattern matching

Pattern matching

> Operation `match` can perform pattern matching on the given argument. Think of it as a `switch` statement on steroids.

Without

```
(define (r:eval-builtin sym)
  (cond [(equal? sym '+) +]
        [(equal? sym '*) *]
        [(equal? sym '-) -]
        [(equal? sym '/) /]
        [else #f]))
```

With `match`

```
(define (r:eval-builtin sym)
  (match sym
    ['+ +]
    ['* *]
    ['- -]
    ['/ /]
    [_ #f]))
```

> The underscore operator `_` means any pattern.

No-match exception

> Operation `match` raises an exception when no pattern is matched, unlike `cond` that returns `#<void>`.

```
(match 1
  [10 #t]) ; Expecting 10, but given 1, so no match
 ; match: no matching clause for 1 [,bt for context]
```

Matching lists

With `cond`

```
(define (factorial n)
  (cond [(= n 0) 1]
        [else (* n (factorial (- n 1)))]))
```

With `match`

Matching lists

With `cond`

```
(define (factorial n)
  (cond [(= n 0) 1]
        [else (* n (factorial (- n 1)))]))
```

With `match`

```
(define (factorial n)
  (match n
    [0 1]
    [_ (* n (factorial (- n 1)))]))
```

Introducing `define/match`

> The define and match pattern is so common that there is a short-hand version. ***Notice the parenthesis!***

With `define/match`

```
(define/match (factorial n)
  [(0) 1]
  [(_) (* n (factorial (- n 1)))])
```

With `match`

```
(define (factorial n)
  (match n
    [0 1]
    [_ (* n (factorial (- n 1)))]))
```

With `cond`

```
(define (factorial n)
  (cond [(= n 0) 1]
        [else (* n (factorial (- n 1)))]))
```

List patterns

- Lists are so common that they deserve a special range of patterns

```
(define (f l)
  (match l
    [(list) #f]           ; Matches the empty list
    [(list 1 2) #t]       ; Matches a list with exactly 1 and 2
    [(list x y) (+ x y)]  ; Matches a list with any two elements
    [(list h t ...) t]))  ; Matches a nonempty list

(check-equal? (f (list)) ???)
(check-equal? (f (list 1) ???)
(check-equal? (f (list 1 2) ???)
(check-equal? (f (list 2 3) ???)
```

List patterns

- Lists are so common that they deserve a special range of patterns

```
(define (f l)
  (match l
    [(list) #f]
    [(list 1 2) #t]
    [(list x y) (+ x y)]
    [(list h t ...) t]))

(check-equal? (f (list)) #f)
(check-equal? (f (list 1) (list))
(check-equal? (f (list 1 2) #t)
(check-equal? (f (list 2 3) (+ 2 3))
```

Example `map`

With `cond`

```
(define (map f l)
  (cond [(empty? l) l]
        [else (cons (f (first l)) (map f (rest l)))]))
```

With `match`

Example `map`

With `cond`

```
(define (map f l)
  (cond [(empty? l) l]
        [else (cons (f (first l)) (map f (rest l)))]))
```

With `match`

```
(define (map f l)
  (match l
    [(list) l]
    [(list h t ...) (cons (f h) (map f t))]))
```

The `#:when` clause

With `match`

```
(define (member x l)
  (match l
    [(list) #f]
    [(list h _ ...) #:when (equal? x h) #t]
    [(list _ t ...) (member x t)]))
```

With `cond`

```
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

- Use the `#:match` clause to add a condition to the pattern

struct patterns

- Match also supports structs

```
(struct foo (bar baz))
(define (f x)
  (match x
    [(foo a b) (+ a b)]))
(check-equal? (f (foo 1 2)) 3)
```

Exercise `r:eval-exp`

With `cond`

```
(define (r:eval-exp exp)
  (cond
    ; 1. When evaluating a number, just return that number
    [(r:number? exp) (r:number-value exp)]
    ; 2. When evaluating an arithmetic symbol, return the respective arithmetic function
    [(r:variable? exp) (r:eval-builtin (r:variable-name exp))]
    ; 3. When evaluating a function call evaluate each expression and apply
    ;    the first expression to remaining ones
    [(r:apply? exp)
     ((r:eval-exp (r:apply-func exp))
      (r:eval-exp (first (r:apply-args exp)))
      (r:eval-exp (second (r:apply-args exp))))]
    [else (error "Unknown expression:" exp)]))
```

Example `r:eval-exp`

```
(define/match (r:eval-exp exp)
  ; 1. When evaluating a number, just return that number
  [((r:number n)) n]
  ; 2. When evaluating an arithmetic symbol, return the respective arithmetic function
  [((r:variable x)) (r:eval-builtin x)]
  ; 3. When evaluating a function call evaluate each expression and apply
  ;    the first expression to remaining ones
  [((r:apply ef (list ea1 ea2))) ((r:eval-exp ef) (r:eval-exp ea1) (r:eval-exp ea2))]
  [(_) (error "Unknown expression:" exp)])
```

Formalism

$$n \Downarrow n \qquad x \Downarrow \mathrm{builtin}(x) \qquad \frac{e_f \Downarrow v_f \qquad e_{a_1} \Downarrow v_{a_1} \qquad e_{a_2} \Downarrow v_{a_2} \qquad v = v_f(v_{a_1}, v_{a_2})}{(e_f\ e_{a_1}\ e_{a_2}) \Downarrow v}$$

Pattern matching

Pros

- Write less code
- Better safety (some languages support exhaustive pattern matching)

Cons

- Exposes your data as public (more maintenance)
- Any changes to your data, breaks patterns that match that data (tighter coupling)

Implementing match

Implementing match for list

```
(define (list-match l on-empty on-cons)
  (cond
    [(empty? l) (on-empty)]
    [(list? l) (on-cons (first l) (rest l))]
    [else (error "Not a list!")]))

(define (length l)
  (list-match l
    (lambda () 0)
    (lambda (_ t) (+ 1 (length t)))))
```

Implementing match for sets of structs

> Racket's `match` is not exhaustive; we do get a runtime error if no branch is met. But how can we know if we are writing all branches?

```
(define (s:value? v)
  (or (s:number? v)
      (s:void? v)
      (s:closure? v)))
(struct s:void () #:transparent)
(struct s:number (value) #:transparent)
(struct s:closure (env decl) #:transparent)
```

We can implement a function that works like match

with fixed branches

Implementing match for sets of structs

```
(define (match-s:value v on-number on-void on-closure)
  (cond [(s:number? v) (on-number (s:number-value v))]
        [(s:void? v) (on-void)]
        [(s:closure? v) (on-closure (s:closure-env v) (s:closure-decl v))]))
; Example:
(define (value-to-id v)
  (match-s:value v
    (lambda (x) 'number)
    (lambda () 'void)
    (lambda (env decl) 'closure)))
```

Pros

- The user **must** provide the code for every case

Cons

- The order of the branches is not easy to remember

Introducing keyword arguments

> We can prefix a function parameter with a `#:symbol` to declare that the order of the arguments does not matter, the name of the parameter does (known as the keyword in Racket).

```
(define (match-s:value v #:number on-number #:void on-void #:closure on-closure)
  (cond [(s:number? v) (on-number (s:number-value v))]
        [(s:void? v) (on-void)]
        [(s:closure? v) (on-closure (s:closure-env v) (s:closure-decl v))]))
; Example:
(define (value-to-id v)
  (match-s:value v
    #:void (lambda () 'void)
    #:number (lambda (x) 'number)
    #:closure (lambda (env decl) 'closure)))
```